# COL 764 : Boolean Retrieval System

Sidharth Agarwal - 2019CS50661

26/08/2021

## Introduction

This program is an implementation of the boolean retrieval system in python. It allows user to test 4 types of compression of the index list, remove stop words, stem the terms and query both single and multi-keywords retrieval, over a collection of xml documents.

## Implementation Details

- ***Parsing Document:*** I used the beautifulsoup library in python for assisting me in parsing and tokenizing the document. I used the standard dictionary of python to store tokens as it's keys and their corresponding posting lists as values, similarly for document id and document name mapping.
- ***Index File:*** The first byte of all the index files contains the type of compression used. The next section of binary file consists of the document names (DOCNO tags here) arranged in increasing order of document index used in posting lists. The last section consists compressed version(if used) of sorted posting list of the tokens.

  The lengths of posting lists is saved first before the posting list so as to distinguish between two posting lists while decompressing. Similarly the number of documents and number of tokens are saved before the start of each section.

  For the mapping of document names to document ids, I didn't use any compression in any case. Since for each character of the name it is possible store it in 1 byte, so there is no point of compressing it in compression 1 because it will take 2 bytes for ascii values greater than 127. Similarly for compression 2 most of the encodings take close to 8 bits or even more(for example value 16 will take 9 bits to be stored in encoded form), and I didn't see any improvement in using that as well.
- ***Dictionary File:*** The dictionary consists the mapping of each token to the location of the byte in the binary file from which we can start reading the length of its posting list, which is followed by the posting list.
- ***External Merging:*** So I used algorithm similar to *Single Pass In Memory Inversion*, where I parsed the documents storing posting lists until I parse 100K documents, if this occurred for the first time, I simply save these posting lists in sorted order in a temporary file on disk and clear the memory. Otherwise I merge the data which was earlier saved with the current posting lists while maintaining the sorted order and appropriate pointers
- ***Retrieval***: For single search retrieval I simply returned the posting list for the token. For multi search retrieval I took the intersection of all the sorted posting lists for each token to get the list containing all the documents having each token, in *O(n1+n2+n3..nm)* where there are m tokens and kth token has a posting list of length nk.
- ***Compression 0***: I didn't use gap encoding, whereas used it for rest others. Here I assumed that the total number of documents available will be less than 4 billion so that I can store each id in 4 bytes. I also thought of a method in C0 where I will store each digit as a 1 byte character and each number separated by a non-digit character, although it gave a better compression for the given collection but it will not be good for a large general number of documents since each digit takes 1 byte, so I discarded this method. I used the *to_bytes* function to repeatedly to convert the integers to bytes before writing them.
- ***Compression 1***: I had split each integer among 7 bits using left shift operators and then right shift as well to put 1 at the 8th bit accordingly, I used a stack to put the bytes generated in the right sequence.

- ***Compression 2***: I did not use any kind of padding on the bits in the encoding given in the problem statement, I wrote bytes such that I considered the bits left from the previous itertion as well and carried the bits which were not divisible by 8 to the next iteration. I took the help of *bin* function avoid loss of 0 bit in this process of carrying to the next iteration
- ***Compression 3***: I used only gap encoding with normal encoding of compression 0, saving it in a temporary file and then compressed it using snappy to get the final compressed indexfile, followed by deleting the temporary file.
- ***Compression 4 Analysis***: So first I observed if we have the number which is to be decoded as x, having a binary representation of length n. Then if I take $b = 2^k$, then my q will be of the order $2^{n-k}$ and so the number of bits to represent U(q+1) will be of order $2^{n-k}$, now since we can have r as $0 \leq r < b$. so r will require k-1 bits for representation. Now the total bits for representing U(q+1).C(r) will be $2^{n-k} + k - 1$ which will have its least value when $k = n$. So now this basically transforms to the method of first storing the value of k which is equal to number of bits of x and then storing its k-1 least significant bits later.
- ***Compression 4 Implementation:*** Now since for optimal k we will always have our q = 1, so there is no point of wasting 2 bits to store U(2) = 10. Now for any number in the first 5 bits, I stored the value of (n-1), where $2^{n-1} < x < 2^n$, since the value of x can be represented in 32 bits. Now in the next $n - 1$ bits I stored the least $n - 1$ significant bits of x. So basically every integer was stored with 4 more bits than the optimal value.
- ***Decompression***: I loaded the complete dictionary and document index mapping(to their names) whereas posting lists for only those tokens which are present in some query in queryfile were decompressed.

# Evaluation Metrics

All the metrics have been evaluated on the document collection given.

## Index Size Ratio

Here D is the size of the dictionary file, P is the size of the postings file, and C is the size of the entire collection used all mentioned in MBs

| Compression Type | D | P | C | ISR |
|---|---|---|---|---|
| 0 | 6.9 | 151.0 | 514.9 | 0.306 |
| 1 | 6.6 | 44.9 | 514.9 | 0.100 |
| 2 | 6.6 | 35.2 | 514.9 | 0.081 |
| 3 | 6.9 | 71.4 | 514.9 | 0.152 |
| 4 | 7.3 | 41.3 | 514.9 | 0.094 |

## Compression Speed

The total time taken to compress all the postings lists, which is the time difference between indexing with a compression strategy and indexing without any compression. The **total time** taken for compression 0 which is used for reference and is also equal to time for indexing, in my case was **681.615 seconds**. Note these values depends on a lot factors of state of computing device like heating, so I would encourage to recheck these.

| Compression Type | Compression Time(in s) |
|---|---|
| 1 | 1.685 |
| 2 | 39.715 |
| 3 | 12.293 |
| 4 | 45.678 |

## Query Speed

Average time taken per query (including decompression of list(s) if required). Here I didn't include the time of loading dictionary file and parsing queries which turned out to be around **0.924 seconds** for all of them. The query time has been averaged over 50 random queries in milliseconds.

| Compression Type | Query Time(in ms) |
|---|---|
| 0 | 36.21 |
| 1 | 42.34 |
| 2 | 132.75 |
| 3 | 50.27 |
| 4 | 93.40 |